

```

1 Punto 1:
2 Representación:
3 fotonica se representa con estr donde
4     estr es tupla ( diccFotos: DiccAVL(foto,
5         tupla ( fecha: tupla(dia: nat, mes: nat, año: nat),
6             itFecha: itLista(foto),
7             tags: Lista(tag), // Recordemos que tener los its en la misma Lista llevaba a problemas
8             listaDeItsTag: Lista(itLista(foto)),
9             ubicación: ubicacion
10         ))
11         fotosPorTag: DiccTrie(tag, Lista(foto))
12         fotosPorFecha: DiccTrie(tupla(dia: nat, mes: nat), Lista(foto) )
13     )
14     foto es nat, ubicacion es nat, tag es string.
15 ---
16 "Aclaración": Con Lista me refiero al "Módulo Lista Enlazada" que se encuentra en el apuntes de Módulos Básicos.
17 Elegí esta estructura para representar las secuencias en todo el parcial debido a que el Módulo mencionado se explica con el TAD
18 • Secuencia.
19 Solución Informal: (explicación charlada de la estructura)
20 - e.fotosPorTag es un diccionario implementado sobre un Trie que posee como clave a todos los tags del sistema, y como significado de
21 • cada tag tiene la lista de fotos que fueron ingresadas con ese Tag.
22 • - Se eligió un Trie ya que sabemos que las operaciones principales tienen un costo en base al largo de la clave, es decir en base
23 • al tag más grande. Pero por el enunciado sabemos que la longitud de los Tags, por lo tanto las complejidades de las operaciones
24 • Definir, Significado y Borrar tendrán una complejidad de  $O(1)$  (sin contar el costo de copiar los datos en Definir).
25 - e.fotosPorFecha es un diccionario implementado sobre un Trie con clave una tupla(dia: nat, mes: nat) y como significado de estás
26 • una lista con todas las fotos que tienen dicha fecha. El hecho de que las claves sean una tupla como la mencionada no es trivial
27 • debido a que se debe explicar como un Trie puede tener tal clave.
28 Supongo un Trie con la siguiente forma:
29 | - |
30 /      /      \
31 |1|    |2|    ... |12|
32 / \    / \    / \
33 |1|..|31| |1|..|31| |1|..|31|
34
35 Con la primer fila de nodos representando a los meses, y la segunda fila representando a los días de cada mes (supongo que los meses
36 • con menos días que 31 solo se usaran hasta el día que les corresponda). Cabe destacar que esta es la forma final del Trie en
37 • realidad, en particular el Trie puede comenzar sin ninguno de los nodos mostrados.
38 Lo importante de este trie es que se puede realizar las operaciones principales en un costo que a priori está acotado por la
39 • naturaleza de los números que se eligieron (ya que el largo máximo de las claves de los tries es a lo sumo 2). De este modo las
40 • operaciones principales tendrán una complejidad de  $O(1)$  (sin contar el costo de copiar los datos en Definir).
41 "Observación": también se podría haber implementado una Matriz para e.fotosPorFecha de dimensión 12 x 31.
42 - e.diccFotos es un diccionario implementado sobre AVL que tiene como clave las fotos y como resultado toda la información de una
43 • foto determinada, es decir sus tags, su ubicación, y su fecha. Además tendrá un iterador (.itFecha) a la posición en la lista de
44 • fotos en donde esté incluida la foto en la estructura de e.fotosPorFecha, además y una lista de iteradores (.listaDeItsTag) donde
45 • cada uno corresponde a una posición por cada una de las listas en donde se encuentra dicha foto en e.fotosPorTag.
46 ---
47 Invariante de Representación:
48 Rep: estr -> boolean
49 (Ve: estr) Rep(e) == true ==> (1)AL(2)AL(3)AL(4)AL(5)AL(6)AL(7)AL(8)AL(9)AL
50 donde:
51 (1) == Las claves del e.diccFotos es igual a la union de las listas de e.fotosPorFecha
52 (2) == La union de las listas de e.fotosPorTag está incluido en las claves del e.diccFotos
53 // (3) == Las fotos de e.fotosPorFecha no se pueden repetir
54 // OBSERVACIÓN: Por lo chequeado en (1) sabemos que no se puede repetir
55 (3) == Todo tag de e.fotosPorTag está definido en los tags de cada foto de e.diccFotos, y viceversa.
56 (4) == Todo tag de toda foto en e.diccFotos debe estar referido en e.diccFotos con la correspondiente foto, y viceversa
57 // (4) Agregado para la solución final
58 (5) == Las tuplas claves de e.fotosPorFecha son fechas validas
59 (6) == Las fechas de cada foto en e.diccFotos son fechas validas
60 // (5)(6) Agregadas para la solución final
61 (7) == Las fotos de las listas de cada fecha de e.fotosPorFecha son fotos en e.diccFotos con tal fecha, y viceversa
62 (8) == El iterador de .itFecha de cada foto en e.diccFotos apunta a la posición donde se encuentra dicha foto en la lista de fotos
63 • de e.fotosPorFecha
64 (9) == La lista de iteradores e.listaDeItsTag de cada foto tiene iteradores a todas las lista de e.fotosPorTag que contienen a
65 • dicha foto con su correspondiente tag.
66
67 donde:
68 tfecha == tupla(dia: nat, mes: nat)
69 (1) == unionDeConjDeSecus(unionDeSignificados(e.fotosPorFecha)) = claves(e.diccFotos)
70 (2) == unionDeConjDeSecus(unionDeSignificados(e.fotosPorTag)) < claves(e.diccFotos)
71 (3) == (Vt: tag)(def?(t, e.fotosPorTag) =>L (E f: foto)(def?(f, e.diccFotos) ^ t E obtener(f, e.diccFotos).tags) ) ^
72 (Vf: foto)(def?(f, e.diccFotos) =>L (E f: tag)(def?(f, e.fotosPorTag) ^ t E obtener(f, e.diccFotos).tags) )
73 (4) == (Vf: foto)(def?(f, e.diccFotos)
74     =>L (Vt: tag) ( t E obtener(f, e.diccFotos).tags =>L está?(f, obtener(t, e.fotosPorTag)) ) ) ^
75 (Vt: tag)(def?(t, e.fotosPorTag)
76     =>L (Vf: foto)( está?(f, obtener(t, e.fotosPorTag)) =>L t E obtener(f, e.diccFotos).tags ) )
77 (5) == (Vtf: tfecha)(def?(tf, e.fotosPorFecha) =>L 1 ≤ tf.dia ≤ 31 ^ 1 ≤ tf.mes ≤ 12 ^ casosEspeciales(tf) )
78 (6) == (Vf: foto)(def?(f, e.diccFotos) =>L
79     1 ≤ obtener(f, e.diccFotos).fecha.dia ≤ 31 ^
80     1 ≤ obtener(f, e.diccFotos).fecha.mes ≤ 12 ^
81     casosEspeciales(obtener(f, e.diccFotos).fecha) )
82 // casosEspeciales es una función que atiende los casos particulares de las fechas (no importan para el scope del ejercicio)
83 (7) == (Vtf: tfecha)(def?(tf, e.fotosPorFecha) =>L
84     ( (Vf: foto)( f E obtener(tf, e.fotosPorFecha) ==>
85         (obtener(f, e.diccFotos).fecha.dia = tf.dia ^
86         obtener(f, e.diccFotos).fecha.mes = tf.mes) ) ) )

```

```

72     (8) ≡ (∀f: foto)(def?(f, e.diccFotos) ⇒L
73         (siguiente(fotoTupla.itFecha) = f ∧
74             ( f ∈ obtener( tupla{fotoTupla.fecha.dia, fotoTupla.fecha.mes}, e.fotosPorFecha) )
75         )
76     )
77     donde fotoTupla ≡ obtener(f,e.diccFotos)
78 (9) ≡ (∀f: foto)(def?(f, e.diccFotos) ⇒L
79     ( (Vit: itLista(foto))(it ∈ obtener(f, e.diccFotos).listaDeItsTag) ⇒L
80         (siguiente(it) = f ∧
81             (∃t: tag)( t ∈ obtener(f, e.diccFotos).tags) ∧ (f ∈ obtener(t, e.fotosPorTag)) )
82         )
83     )
84 )
85 - Operaciones Auxiliares
86 unionDeSignificados:   dicc(α, β) d   -> conj(β)
87 unionDeSignificados(d) ≡ if ∅?(claves(d)) then
88     ∅
89     else
90         obtener(dameUno(claves(d)), d) U unionDeSignificados(borrar(dameUno(claves(d))))
91     fi
92 unionDeConjDeSecus: conj(secu(σ)) cs   -> conj(σ)
93 unionDeConjDeSecus(cs) ≡ if ∅?(cs) then
94     ∅
95     else
96         secuAConj(dameUno(cs)) U unionDeConjDeSecus(sinUno(cs))
97     fi
98 secuAConj: secu(σ) s   -> conj(σ)
99 secuAConj(s) ≡ if ∅?(s) then
100     ∅
101     else
102         {prim(s)} U secuAConj(fin(s))
103     fi
104 ---
105 Función de abstracción:
106 Abs:  estr   -> fotónica   {Rep(e)}
107 (Ve: estr) Abs(e) =obs s: fotónica | (1) ∧ (2) ⇒ (3) ∧ (4) ∧ (5))
108 donde:
109 (1) ≡ fotos(s) = claves(e.diccFotos)
110 (2) ≡ (∀f: foto)(fotoRegistrada?(f,s)
111 (3) ≡ tagsPorFoto(f,s) = obtener(f,diccFotos).tags
112 (4) ≡ fechaFoto(f,s) = (obtener(f,diccFotos).fecha.dia,
113                         obtener(f,diccFotos).fecha.mes,
114                         obtener(f,diccFotos).fecha.año)
115 (5) ≡ ubicacionFoto(f,s) = obtener(f,diccFotos).loc
116 ---
117 //-----
118 - Punto 3:
119 Interfaz:
120 Algoritmos del módulo:
121 iAgregarFoto(in/out e: estr, in id: foto, in día: nat, in mes: nat, in año: nat, in lugar: ubicación, in tags: lista(tag) )
122 //----- Sección para actualizar e.fotosPorFecha
123 itLista <- NULL // Donde escriba "(algo) <- NULL" me refiero a que "inicializo la variable"
124 listaFotosDelDia <- NULL
125 if DiccTrie::Definido?(e.fotosPorFecha, tupla(día,mes)) then //0(1)
126     listaFotosDelDia <- DiccTrie::Significado(e.fotosPorFecha, tupla(día,mes)) //0(1)
127     itLista <- Lista::AgregarAdelante(listaFotosDelDia, id) //0(1)
128 else
129     listaNueva <- Lista::Vacía() //0(1)
130     itLista <- Lista::AgregarAdelante(listaNueva, id) //0(1)
131     DiccTrie::Definir(e.fotosPorFecha, tupla(día,mes), listaNueva) //0(1)
132 end if
133
134 //----- Sección para actualizar e.fotosPorTag
135 // OBS: Supongo que la lista Tags no tiene repetidos
136 // Esto es necesario aclararlo, en caso de suponer lo contrario hay que poder resolverlo
137
138 listaDeIts <- Lista::Vacía() //0(1)
139 it <- Lista::CrearIt(tags) //0(1)
140 listaFotos <- NULL
141 while Lista::HaySiguiente(it) do //--0(t)
142     tag <- Lista::Siguiente(it) //0(1)
143
144     if DiccTrie::Definido?(e.fotosPorTag, tag) then //0(1)
145         listaFotos <- DiccTrie::Significado(e.fotosPorTag, tag) //0(1)
146     else
147         listaFotos <- Lista::Vacío()
148         DiccTrie::Definir(e.fotosPorTag, tag, listaFotos) //0(1)
149     end if
150     itListaTags <- AgregarAdelante(listaFotos, id) //0(1)
151     Lista::AgregarAdelante(listaDeIts, itListaTags) //0(1)
152     Lista::Avanzar(it) //0(1)
153 end while
154
155
156
157

```

```

158 //----- Sección para actualizar e.diccFotos
159 DiccAVL::Definir(e.diccFotos, id,
160     tupla( tupla (dia,mes,anio), itLista, tags, listaDeIts, lugar  )
161     ) //  $O(\log(n) + \text{copy}(\text{tags})) = O(\log(n) + t)$ 
162 // Recordamos que cambie la solución de la clase para no tener la lista de Tags pegada a la lista de sus iteradores.
163 ---
164 Justificación de Complejidad:
165 - Sección para actualizar e.fotosPorFecha: se tiene  $O(1)$  porque todas las operaciones fueron hechas sobre el Trie acotado por las
166 fechas
167 - Sección para actualizar e.fotosPorTag: el costo de las operaciones internas del ciclo también están beneficiadas por tener la
168 longitud máxima de los tags acotada, por lo tanto se cuenta la complejidad del ciclo solo como  $O(t)$  que es el costo de haber
169 recorrido todos los tags haciendo operaciones  $O(1)$  por cada iteración.
170 - Sección para actualizar e.diccFotos: Solo se cuenta el costo del Definir, lo cual es  $O(\log(n) + \text{copy}(\text{clave}) +$ 
171  $\text{copy}(\text{significado}))$ , donde  $\text{copy}(\text{clave}) = \text{copy}(\text{id}) = O(1)$  porque id es un Nat. Y el costo de copiar el significado se reduce a
172 copiar muchos datos primitivos sumado a la lista de tags y una lista de iteradores de igual tamaño. Por lo tanto se tiene una
173 complejidad de  $O(\log(n) + t + t) = O(\log(n) + t)$ 
174
175 Finalmente llegamos a la complejidad final de  $O(1 + t + \log(n) + t) = O(\log(n) + t)$ .
176
177 //-----
178 - Punto 2:
179 - BorrarFoto:
180
181 Interfaz:
182 Algoritmos del módulo:
183 iBorrarFoto(in/out e: estr, in id: foto)
184 //----- Sección para actualizar e.diccFotos
185 tuplaFoto <- DiccTrie::Significado(e.diccFotos, id) //  $O(\log(n) + ts)$ 
186 DiccTrie::Borrar(e.diccFotos, id) //  $O(\log(n))$ 
187
188 //----- Sección para actualizar e.fotosPorFecha
189 // OBS: No creo necesario borrar la clave porque son dias del año
190 Lista::EliminarSiguiente(tuplaFoto.itFecha) //  $O(1)$ 
191
192 //----- Sección para actualizar e.fotosPorTag
193 it <- Lista::CrearIt(tuplaFoto.tags) //  $O(1)$ 
194 while Lista::HaySiguiente(it) do //  $O(ts)$ 
195     itParaBorrar <- Lista::Siguiente(it).itTag //  $O(1)$ 
196     Lista::EliminarSiguiente(itParaBorrar) //  $O(1)$ 
197     Lista::Avanzar(it) //  $O(1)$ 
198 end while
199
200 // Código equivalente del While anterior
201 // (solo para mostrar un ejemplo de como podrían hacerlo con un For)
202 //-----
203 for i ← 0, i < Lista::Tamaño(tuplaFoto.tags), i++
204     itParaBorrar <- tuplaFoto.tags[i].itTag
205     Lista::EliminarSiguiente(itParaBorrar)
206 end for
207 //-----
208 ---
209 Justificación de Complejidad:
210 - Sección para actualizar e.diccFotos: tanto buscar como borrar en un AVL de estas características se realiza en  $O(\log(n))$ .
211 - Sección para actualizar e.fotosPorFecha: como se guardo un iterador se puede borrar en  $O(1)$ .
212 - Sección para actualizar e.fotosPorTag: se tiene un ciclo que recorre la lista de tamaño t, la cual contiene los iteradores para
213 borrar las fotos. Y como las operaciones internas solo hacen trabajo de iteradores, las 3 líneas cuestan  $O(1)$ . Por lo tanto el
214 ciclo cuesta  $O(t)$ . Hace falta aclarar que  $O(t)$  es mejor que  $O(ts)$  que fue lo pedido en el enunciado porque  $t \leq ts$ . De esta forma
215 se llega a la complejidad pedida por el enunciado  $O(ts)$ 
216
217 Finalmente llegamos a la complejidad final de  $O(\log(n) + 1 + t) = O(\log(n) + t) = O(\log(n) + ts)$ .
218
219 - Foto Registrada:
220 Interfaz:
221 Algoritmos del módulo:
222 iFotoRegistrada?(in e: estr, in id: foto, out res: bool)
223 res <- DiccAVL::Definido?(e.diccFotos, id) //  $O(\log(n))$ 
224 ---
225 Justificación de Complejidad: La complejidad final es  $O(\log(n))$  por tratarse de un AVL, y además porque se devuelve un tipo de dato
226 primitivo.
227
228 - TagsPorFoto:
229 Interfaz:
230 Algoritmos del módulo:
231 iTagsPorFoto(in e: estr, in id: foto, out res: lista(tag))
232 res <- DiccTrie::Significado(e.diccFotos, id).tags //  $O(\log(n))$  devolviendo ref no modificable
233 //Recordemos por última vez que a diferencia de la clase, modifique la estructura para que la lista de tags no esté
234 // unida a la lista de iteradores, con lo cual esta operación se podía hacer fácil y en la complejidad adecuada.
235 ---
236 Justificación de Complejidad: La complejidad final es  $O(\log(n))$  por tratarse de un AVL. Además devuelvo una referencia no modificable, de
237 esa forma me guardo el costo de realizar una copia de la lista (lo cual sería  $O(n)$ ).
238
239
240
241
242
243
244
245

```

```

233 - FotosPorTag:
234 Interfaz:
235 Algoritmos del módulo:
236   iFotosPorTag(in e: estr, in tag: tag, out res: lista(foto))
237   res <- DiccTrie::Significado(e.fotosPorTag, tag) // O(1) devolviendo ref no modificable
238   ---
239 Justificación de Complejidad: Devolver un significado de un Trie que tiene acotadas las claves es O(1). Sumado a que se devuelve una
  * referencia no modificable obtenemos O(1).
240
241 - FotosDelDia:
242 Interfaz:
243 Algoritmos del módulo:
244   iFotosDelDia(in e: estr, in día: nat, in mes: nat, out res: lista(foto))
245   res <- DiccTrie::Significado(e.fotosPorFecha, tupla(día,mes)) // O(1) devolviendo ref no modificable
246   ---
247 Justificación de Complejidad: Devolver un significado de un Trie que tiene acotadas las claves es O(1). Sumado a que se devuelve una
  * referencia no modificable obtenemos O(1).
248
249
250 //-----
251 Punto 4 (Opcional):
252 Se tiene que realizar el siguiente cambio en la estructura:
253 Representación:
254 tipoDato se representa con estr donde
255 estr es tupla (diccFotos: DiccAVL(foto,
256                               tupla(fecha: tupla(día: nat, mes: nat, año: nat),
257                                     itFecha: itlista(foto)
258                                     tags: lista(tag)
259                                     listaDeItsTag: lista(itlista(foto))
260                                     ubicacion: ubicacion
261                               ),
262                               fotosPorTag: DiccTrie(tag, lista(foto)) )
263                               fotosPorFecha: DiccTrie(tupla(día: nat, mes: nat), lista(foto)),
264                               ubicacionesMásFotografias: MaxHeap(tupla(cant: nat, ubicaciones: lista(ubicacion))
265                               ---
266
267 Solución Informal:
268 - Agregó un MaxHeap a la estructura, el cual va a estar ordenado por el parametro .cant de su tupla. Y la idea es que este el
  * criterio de orden sea dado por tal parametro justamente. Y para cada una de las cantidad del MaxHeap tendremos la lista de
  * ubicaciones que tienen esa cantidad de apariciones. (Aclaro que uso un MaxHeap normal, y no un Fibonacci MaxHeap).
269
270 Ahora puedo pasar a justificar las complejidades nuevas de cada operación.
271 - ubicacionMásFotografiada(in e: estr, out res: ubicacion)
272 Sé que se puede obtener el máximo elemento de un MaxHeap en O(1), por lo tanto la complejidad de la operación es mejor que la pedida
  * por el enunciado.
273 - AgregarFoto(in/out e: estr, in id: foto, in día: nat, in mes: nat, in año: nat, in lugar: ubicacion, in tags: lista(tag) )
274 La complejidad anterior era: O(log(n) + t).
275 (Aclaración: Se puede calcular las complejidades del Heap en base al parametro "u" ya que en el peor caso la cantidad de claves del
  * heap pueden ser iguales a "u" justamente)
276 Ahora debemos agregar la complejidad de actualizar el heap, es decir actualizar la ubicacion de la foto que está entrando. Para ello
  * tenemos 3 casos:
277 - La ubicación no existe: Por lo tanto se la debe agregar al MaxHeap como una tupla con .cant = 1, lo cual cuesta O(log(u))
278 - La ubicación era la más fotografiada: Significa que está en el tope del heap, e incrementar en 1 a este elemento es O(1), ya
  * que si este ya era el máximo entonces incrementar en uno el máximo no afecta al heap.
279 - La ubicación no era la más fotografiada: En este caso si va a haber cierto movimiento dentro de la estructura debido a que se
  * tiene que incrementar en 1 alguna de las claves. Esta operación es común en los heaps y hasta tiene su propio nombre: increase-
  * key, el cual en el caso general es O(log(u)), pero lamentablemente no podríamos usarla con esa complejidad. Esto se debe a que
  * cada clave hace referencia a una lista de ubicaciones, de la cual se tiene que buscar y sacar la ubicación a modificar, y hacer
  * esta búsqueda cuesta O(u) en el peor caso. Por lo tanto la complejidad de este caso es O(u). (Esto se puede mejorar facilmente
  * guardando un iterador a la posición de la lista donde esté la ubicación, pero quería dejar una solución sencilla a modo de
  * ejemplo).
280 Por lo tanto el costo de modificar la estructura en el peor caso es O(u), dejandonos una complejidad final de O(log(n) + t + u)
281 - BorrarFoto(in/out e: estr, in id: foto)
282 La complejidad anterior era: O(log(n) + ts).
283 Solo debemos analizar la modificación a realizar sobre el heap, por lo tanto vamos a separar en casos:
284 - La foto borrada tenía la ubicación más fotografiada: En el caso general de un heap se puede usar la operación contraria a la
  * que mencione anteriormente llamada decrease-key que es O(log(u)), pero surge el mismo problema mencionado antes, por lo cual nos
  * va a quedar una complejidad de O(u) en el peor caso.
285 - La foto borrada NO tenía la ubicación más fotografiada: esto a su vez puede dividirse en dos casos:
286 - La ubicación no se tiene que borrar: Significa que la lista de ubicaciones no estaba vacía, por lo que se tendrá la misma
  * situación de antes, que resultaba en O(u).
287 - La ubicación se tiene que borrar: Significa que para una clave del heap se debería quedar con una lista vacía al borrar la
  * foto, lo cual significa que tal clave debe ser borrada lo cual con una implementación naive se borra de un MaxHeap en O(u).
  * (Se puede borrar en O(log(u)) pero el algoritmo no es nada trivial).
288 Para resumir, modificar el heap nos cuesta O(u) en el peor caso. Esto deja una complejidad final de O(log(n) + ts + u)
289 - El resto de las operaciones no se ven afectadas por el cambio realizado.
290
291 Para concluir, nos quedan las siguientes complejidades:
292 -ubicacionMásFotografiada(): O(1)
293 -AgregarFoto(): O(log(n) + t + u)
294 -BorrarFoto(): O(log(n) + ts + u)
295

```